What is SQL Injection ?

SQL Injection is a code injection technique used to execute malicious / unauthorized SQL Statements.

Working of SQL Injection:

Eg. Login into the web application.

Username	abc@xyz.com
Password	123456

Username	**Password**
abc@xyz.com	123456
Ajeet	1234
BridgeSoft	123654

SQL Query:

Select * from users where username = ‘abc@xyz.com’ and password ‘123456’.

Return Login or invalid login.

Manipulated SQL Query Injection :

To do with OR- Logic Gate.

A	B	Out
False	False	False
True	False	True
False	True	True
True	True	True

Select * from users where username = ‘’ OR 1=1--’

First ‘ is to close the string parameter then there is a OR function after this a statement is 1=1

The statement **1=1** is always true . As per the OR gate If one of the statements is true then the result becomes true hence the SQL query always returns true.

Processing SQL Statement with JDBC.
1. Establishing a connection.
2. Create a Statement .
3. Execute the query.
4. Process the ResultSet object.
5. Close the connection.

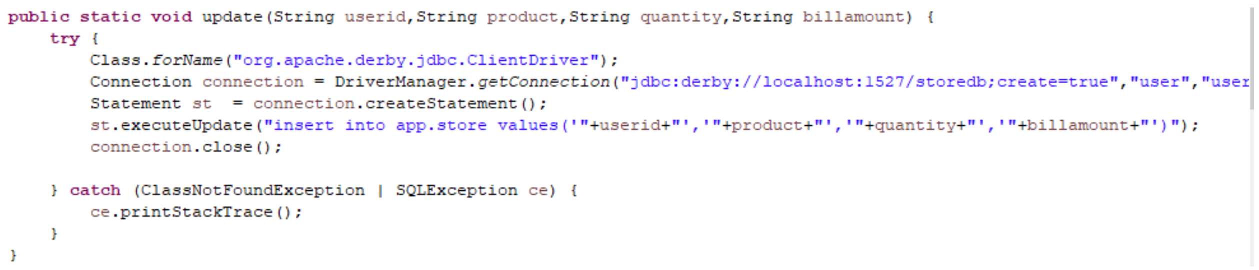
2. Creating Statement

A Statement is an interface that represents a SQL statement. You execute Statement objects, and they generate ResultSet objects, which is a table of data representing a database result set. You need a Connection object to create a Statement object.

Ex: stmt = con.createStatement();

There are three different kinds of statements:

- Statement: Used to implement simple SQL statements with no parameters.



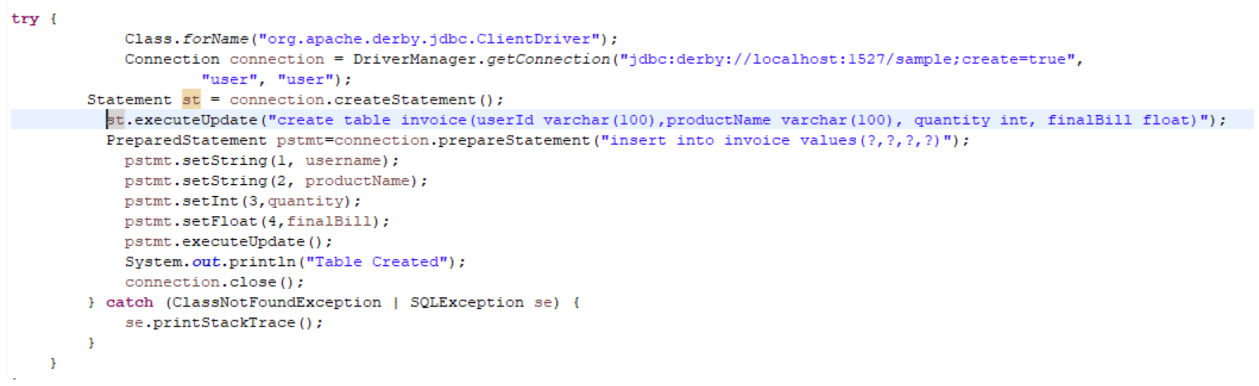
```
public static void update(String userid,String product,String quantity,String billamount) {
    try {
        Class.forName("org.apache.derby.jdbc.ClientDriver");
        Connection connection = DriverManager.getConnection("jdbc:derby://localhost:1527/storedb;create=true","user","user
        Statement st  = connection.createStatement();
        st.executeUpdate("insert into app.store values('"+userid+"','"+product+"','"+quantity+"','"+billamount+"')");
        connection.close();

    } catch (ClassNotFoundException | SQLException ce) {
        ce.printStackTrace();
    }
}
```

- **PreparedStatement: (Extends Statement.)**

  Used for precompiling SQL statements that might contain input parameters. The main feature of a PreparedStatement object is that, unlike a Statement object, it is given a SQL statement when it is created. The advantage to this is that in most cases, this SQL statement is sent to the DBMS right away, where it is compiled. As a result, the PreparedStatement object contains not just a SQL statement, but a SQL statement that has been precompiled. This means that when the PreparedStatement is executed, the

DBMS can just run the PreparedStatement SQL statement without having to compile it first.

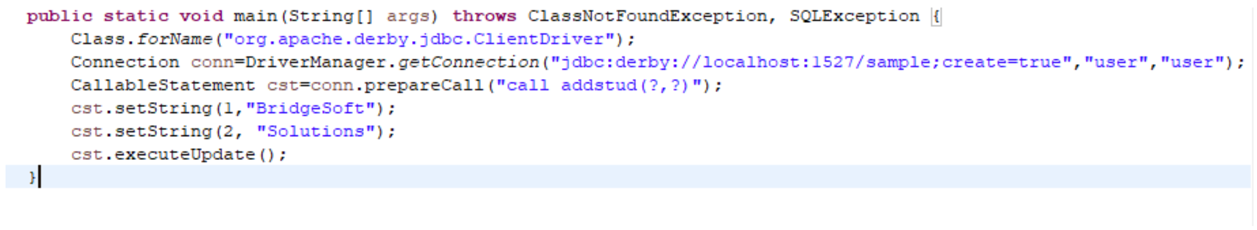
Although you can use PreparedStatement objects for SQL statements with no parameters, you probably use them most often for SQL statements that take parameters. The advantage of using SQL statements that take parameters is that you can use the same statement and supply it with different values each time you execute it. Examples of this are in the following sections.



```
try {
            Class.forName("org.apache.derby.jdbc.ClientDriver");
            Connection connection = DriverManager.getConnection("jdbc:derby://localhost:1527/sample;create=true",
                    "user", "user");
        Statement st = connection.createStatement();
          st.executeUpdate("create table invoice(userId varchar(100),productName varchar(100), quantity int, finalBill float)");
          PreparedStatement pstmt=connection.prepareStatement("insert into invoice values(?,?,?,?)");
            pstmt.setString(1, username);
            pstmt.setString(2, productName);
            pstmt.setInt(3,quantity);
            pstmt.setFloat(4,finalBill);
            pstmt.executeUpdate();
            System.out.println("Table Created");
            connection.close();
        } catch (ClassNotFoundException | SQLException se) {
            se.printStackTrace();
        }
    }
```

- **CallableStatement: (Extends PreparedStatement.)**

  Used to execute stored procedures that may contain both input and output parameters.



```
public static void main(String[] args) throws ClassNotFoundException, SQLException {
    Class.forName("org.apache.derby.jdbc.ClientDriver");
    Connection conn=DriverManager.getConnection("jdbc:derby://localhost:1527/sample;create=true","user","user");
    CallableStatement cst=conn.prepareCall("call addstud(?,?)");
    cst.setString(1,"BridgeSoft");
    cst.setString(2, "Solutions");
    cst.executeUpdate();
}
```

Inner Class:

Inner class means one class which is a member of another class. There are basically four types of inner classes in java.

1) Nested Inner class

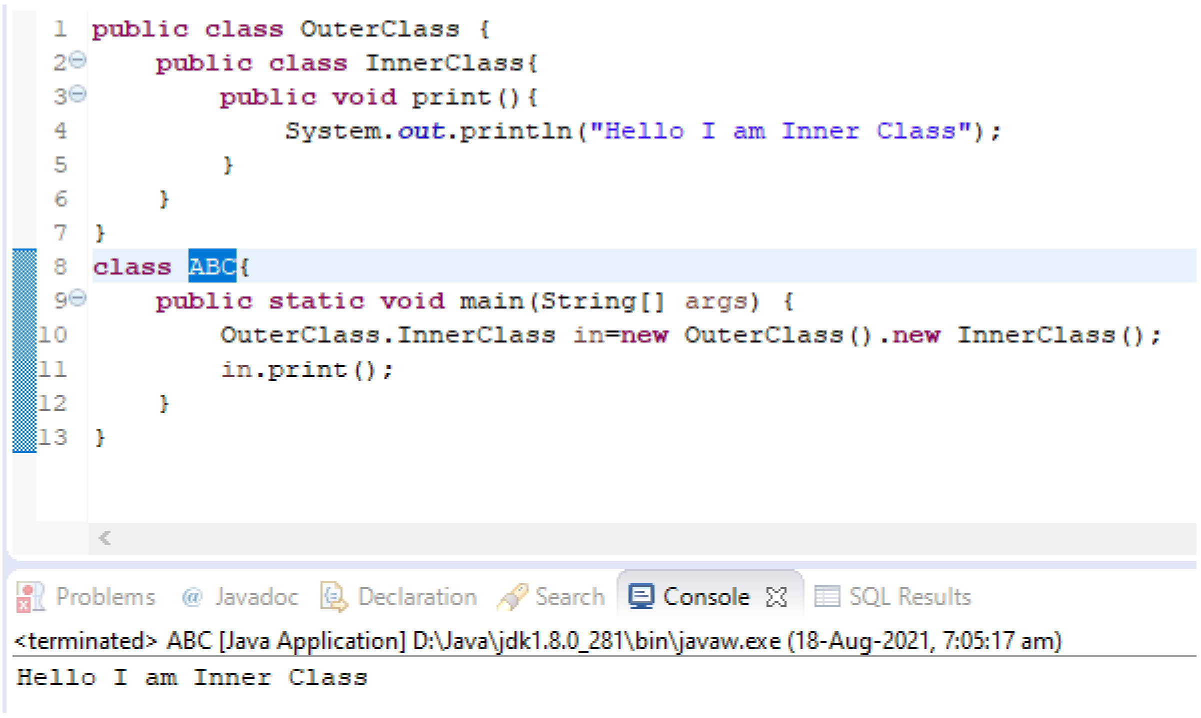
2) Method Local inner classes

3) Anonymous inner classes

4) Static nested classes

1. **Nested Inner class** : This type of inner class involves the nesting of a class inside another class. The inner class can access the private variables of the outer class.

   We can modify access to the inner class by using access modifier keywords such as private, protected, and default. Similarly, access specifiers can help nest interfaces inside one another.

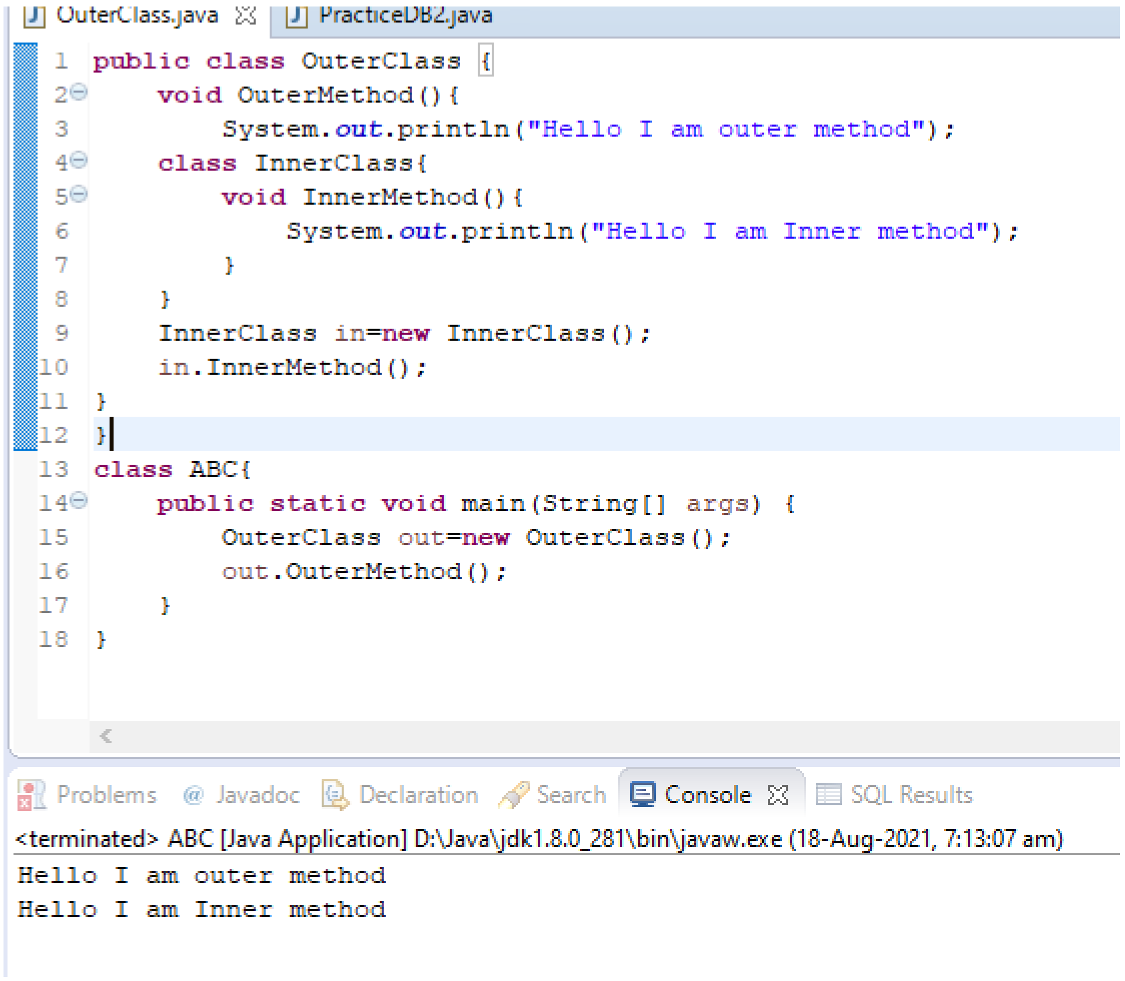


```
public class OuterClass {
    public class InnerClass{
        public void print(){
            System.out.println("Hello I am Inner Class");
        }
    }
}
class ABC{
    public static void main(String[] args) {
        OuterClass.InnerClass in=new OuterClass().new InnerClass();
        in.print();
    }
}
```

```
<terminated> ABC [Java Application] D:\Java\jdk1.8.0_281\bin\javaw.exe (18-Aug-2021, 7:05:17 am)
Hello I am Inner Class
```

2. Method Local Inner Class:

   The inner class cannot use the variables of the outer class if they are not declared as final values. This rule was prevalent until JDK 1.7. After that, inner classes can access non-final local variables.

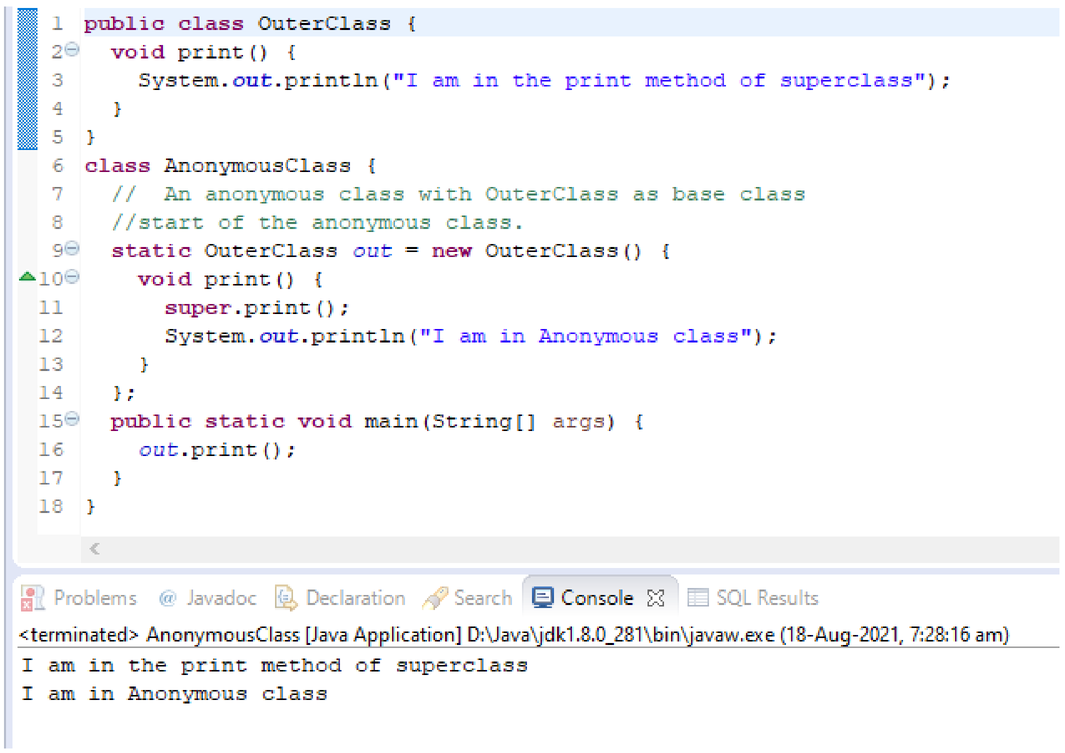

OuterClass.java   PracticeDB2.java

```java
public class OuterClass {
    void OuterMethod(){
        System.out.println("Hello I am outer method");
    class InnerClass{
        void InnerMethod(){
            System.out.println("Hello I am Inner method");
        }
    }
    InnerClass in=new InnerClass();
    in.InnerMethod();
}
}
class ABC{
    public static void main(String[] args) {
        OuterClass out=new OuterClass();
        out.OuterMethod();
    }
}
```

```
<terminated> ABC [Java Application] D:\Java\jdk1.8.0_281\bin\javaw.exe (18-Aug-2021, 7:13:07 am)
Hello I am outer method
Hello I am Inner method
```

3. Anonymous Inner Class: As per the name suggests these inner classes have no name at all . The definition of these classes is written outside the scope of the outer class. There are of two types:

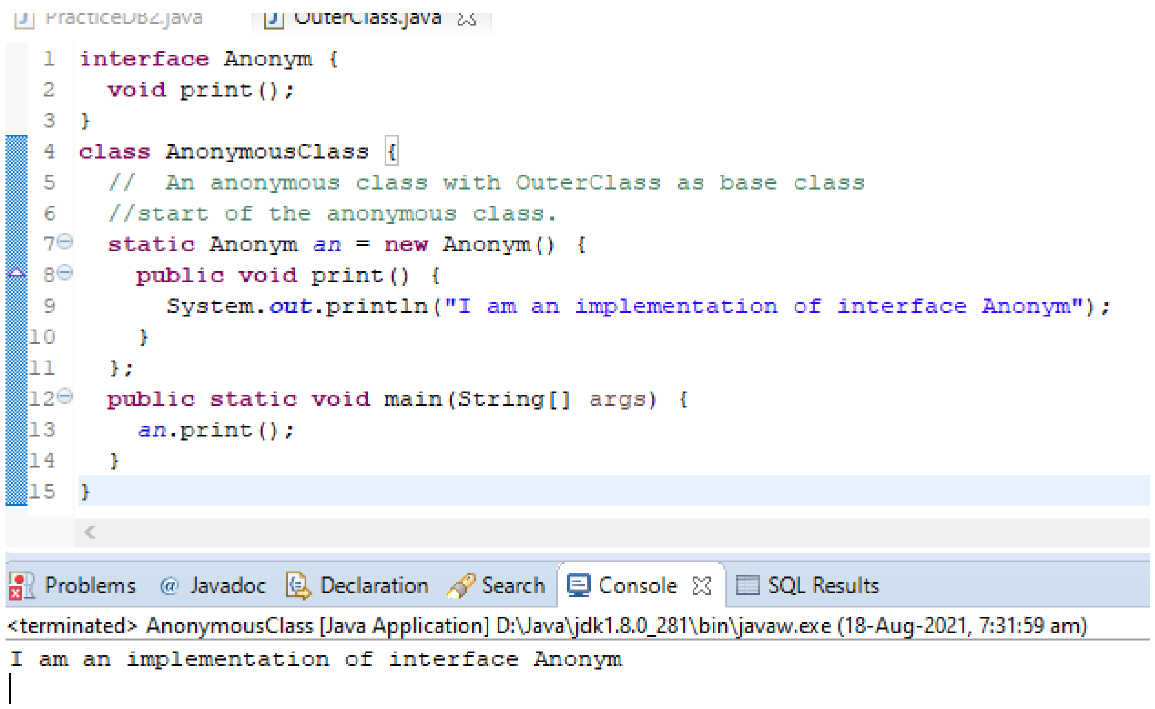
    a. Subclass of the specified types: In this method, the anonymous class is put inside a subclass of the outer class.



```java
public class OuterClass {
  void print() {
    System.out.println("I am in the print method of superclass");
  }
}
class AnonymousClass {
  //  An anonymous class with OuterClass as base class
  //start of the anonymous class.
  static OuterClass out = new OuterClass() {
    void print() {
      super.print();
      System.out.println("I am in Anonymous class");
    }
  };
  public static void main(String[] args) {
    out.print();
  }
}
```

```
<terminated> AnonymousClass [Java Application] D:\Java\jdk1.8.0_281\bin\javaw.exe (18-Aug-2021, 7:28:16 am)
I am in the print method of superclass
I am in Anonymous class
```

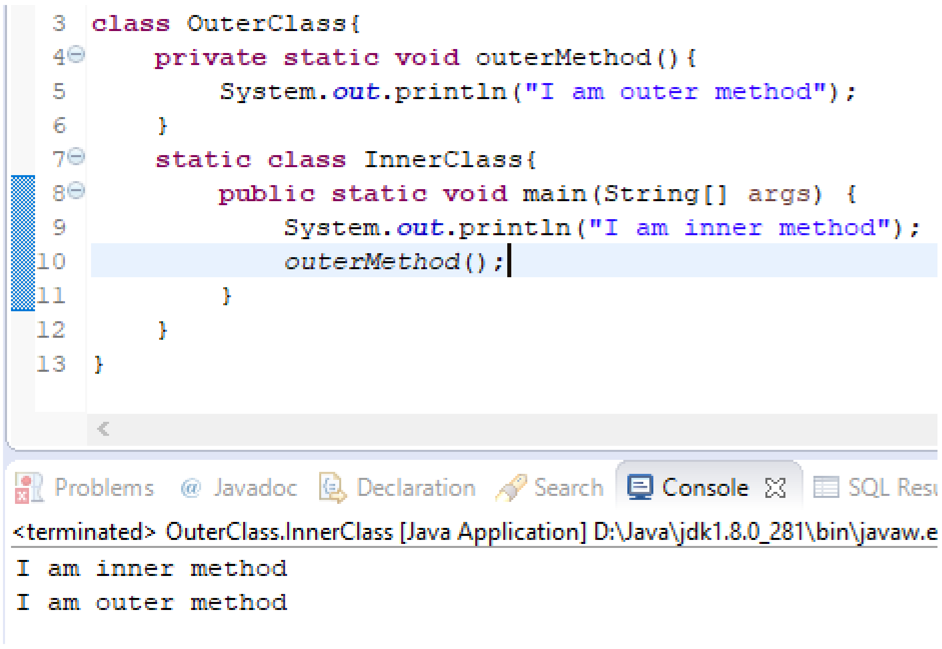
b. The implementer of specified Interface: The anonymous class can extend a class or implement an interface at a time. Here, we will see interface implementation by anonymous classes.



```java
interface Anonym {
  void print();
}
class AnonymousClass {
  //  An anonymous class with OuterClass as base class
  //start of the anonymous class.
  static Anonym an = new Anonym() {
    public void print() {
      System.out.println("I am an implementation of interface Anonym");
    }
  };
  public static void main(String[] args) {
    an.print();
  }
}
```

```
<terminated> AnonymousClass [Java Application] D:\Java\jdk1.8.0_281\bin\javaw.exe (18-Aug-2021, 7:31:59 am)
I am an implementation of interface Anonym
```

4. Static nested classes: They are like static members of the outer class.



```java
class OuterClass{
    private static void outerMethod(){
        System.out.println("I am outer method");
    }
    static class InnerClass{
        public static void main(String[] args) {
            System.out.println("I am inner method");
            outerMethod();
        }
    }
}
```

```
<terminated> OuterClass.InnerClass [Java Application] D:\Java\jdk1.8.0_281\bin\javaw.e
I am inner method
I am outer method
```

Relationship in JAVA

There are three types of relationship in Java

1. Uses - A relationship: This relationship occurs when an object is created inside a method of another class. It is better to make sure that there's less dependency on one another.
   Example: College uses the faculties.

   Syntax:
   ```
   Class A{

   }
   Class B{
           void method(){
                   A obj=new A();
   }
   }
   ```

2. Has A Relation: This relationship occurs when an object of a class is created as a data member of another class. It is easier to understand. Example: Car has an engine.

   Syntax:
   ```
   class A{

   }
   Class B{
   A obj=new A();
   }
   ```

3. Is -A Relationship : This relationship occurs between two classes in which one class extends another class.
   Example: Maruti is a Car.
   Syntax:
   ```
   Class A{

   }
   Class B extends A{

   }
   ```